

# Laboratoire de la semaine 2 (29 février)

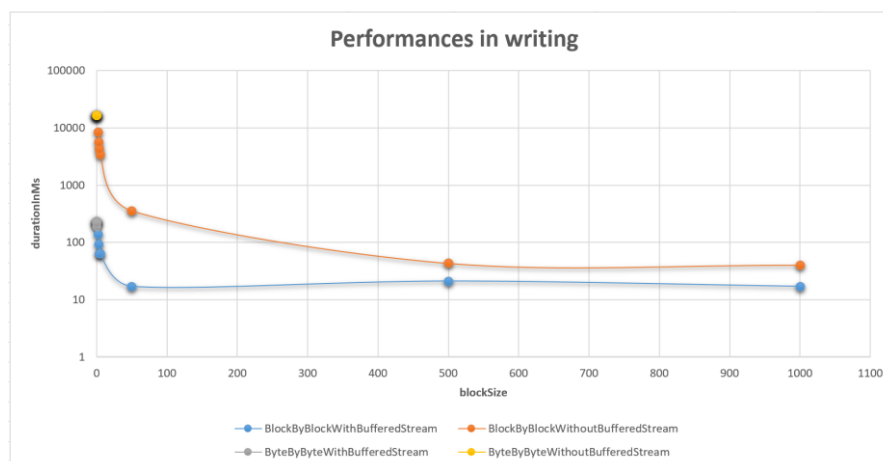
## Conditions de l'expérience

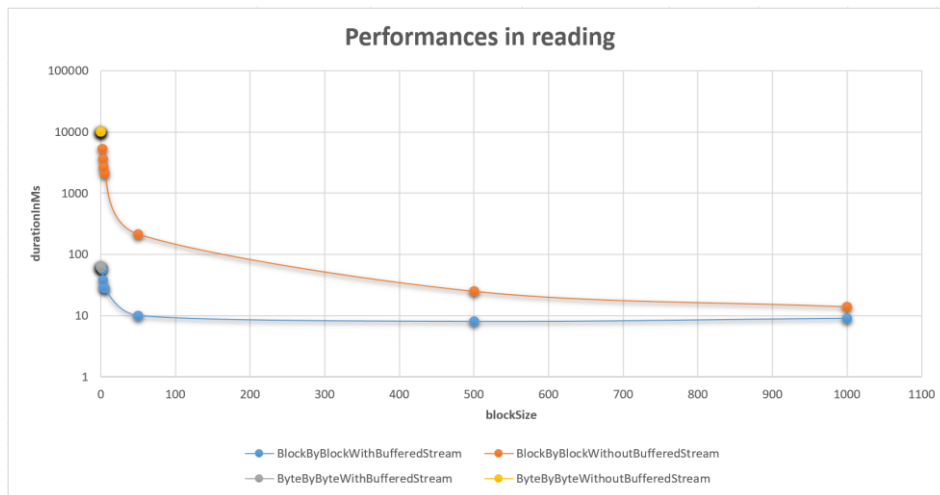
Cette expérience cherche à montrer que les performances peuvent être grandement améliorées lors de l'écriture et de la lecture de données en Java en utilisant simplement des tampons (« buffers » en anglais). Pour le déroulement de l'expérience, nous utilisons un programme écrit en Java (fichier « BufferedIOBenchmark.java ») qui va nous permettre d'écrire des données de tests dans différents fichiers et ensuite de relire les données de ces mêmes fichiers. Nous mettons ainsi en évidence le temps nécessaire pour l'écriture et la lecture de ces données suivant différentes stratégies. Une première stratégie consiste à utiliser des tampons avec une taille de bloc définie (nombre d'octets par bloc), une deuxième consiste à utiliser des tampons mais avec une taille de bloc égale à 0 (les données sont écrites et lues octet après octet), la troisième et quatrième stratégies consistent en la même chose que les deux premières mais en n'utilisant aucun tampon. L'expérience permet ainsi de montrer que l'écriture et la lecture des données est bien plus rapide en utilisant des tampons. L'ordinateur de test utilisé pour l'expérience possède les caractéristiques importantes suivantes : le système d'exploitation Microsoft Windows 10 Professionnel (64 bits), un processeur Intel Core i7-4810MQ de 2.80 GHz, 16 Go de mémoire physique installée (RAM) et un SSD Samsung MZ7TE512HMHP-000L1 de 476.94 Go de capacité.

## Présentation des mesures

Notre programme permet également d'enregistrer dans un fichier au format « csv » les différents temps obtenus pour toutes les stratégies selon l'écriture et la lecture des données. Ci-dessous, nous avons les deux tableaux contenus dans le fichier « csv » généré ainsi que les deux graphiques, se rapportant aux deux tableaux, que nous avons dessinés en utilisant un tableur (Excel). Il est important de préciser que l'échelle utilisée pour représenter la durée en ms est logarithmique (en base 10) pour avoir de meilleurs graphiques. De plus, la taille du fichier dans lequel les données sont écrites est fixée et ne change pas (pour pouvoir effectuer les comparaisons).

operation	strategy	blockSize	fileSizeInBytes	durationInMs	operation	strategy	blockSize	fileSizeInBytes	durationInMs
WRITE	BlockByBlockWithBufferedStream	1000	10485760	17	READ	BlockByBlockWithBufferedStream	1000	10485760	9
WRITE	BlockByBlockWithBufferedStream	500	10485760	21	READ	BlockByBlockWithBufferedStream	500	10485760	8
WRITE	BlockByBlockWithBufferedStream	50	10485760	17	READ	BlockByBlockWithBufferedStream	50	10485760	10
WRITE	BlockByBlockWithBufferedStream	5	10485760	63	READ	BlockByBlockWithBufferedStream	5	10485760	28
WRITE	BlockByBlockWithBufferedStream	4	10485760	64	READ	BlockByBlockWithBufferedStream	4	10485760	29
WRITE	BlockByBlockWithBufferedStream	3	10485760	93	READ	BlockByBlockWithBufferedStream	3	10485760	38
WRITE	BlockByBlockWithBufferedStream	2	10485760	137	READ	BlockByBlockWithBufferedStream	2	10485760	57
WRITE	ByteByByteWithBufferedStream	0	10485760	188	READ	ByteByByteWithBufferedStream	0	10485760	65
WRITE	ByteByByteWithBufferedStream	0	10485760	228	READ	ByteByByteWithBufferedStream	0	10485760	65
WRITE	ByteByByteWithBufferedStream	0	10485760	233	READ	ByteByByteWithBufferedStream	0	10485760	63
WRITE	ByteByByteWithBufferedStream	0	10485760	222	READ	ByteByByteWithBufferedStream	0	10485760	62
WRITE	ByteByByteWithBufferedStream	0	10485760	217	READ	ByteByByteWithBufferedStream	0	10485760	62
WRITE	BlockByBlockWithoutBufferedStream	1000	10485760	40	READ	BlockByBlockWithoutBufferedStream	1000	10485760	14
WRITE	BlockByBlockWithoutBufferedStream	500	10485760	43	READ	BlockByBlockWithoutBufferedStream	500	10485760	25
WRITE	BlockByBlockWithoutBufferedStream	50	10485760	355	READ	BlockByBlockWithoutBufferedStream	50	10485760	214
WRITE	BlockByBlockWithoutBufferedStream	5	10485760	3440	READ	BlockByBlockWithoutBufferedStream	5	10485760	2106
WRITE	BlockByBlockWithoutBufferedStream	4	10485760	4298	READ	BlockByBlockWithoutBufferedStream	4	10485760	2635
WRITE	BlockByBlockWithoutBufferedStream	3	10485760	5684	READ	BlockByBlockWithoutBufferedStream	3	10485760	3642
WRITE	BlockByBlockWithoutBufferedStream	2	10485760	8454	READ	BlockByBlockWithoutBufferedStream	2	10485760	5297
WRITE	ByteByByteWithoutBufferedStream	0	10485760	16450	READ	ByteByByteWithoutBufferedStream	0	10485760	10122
WRITE	ByteByByteWithoutBufferedStream	0	10485760	16450	READ	ByteByByteWithoutBufferedStream	0	10485760	10122
WRITE	ByteByByteWithoutBufferedStream	0	10485760	16599	READ	ByteByByteWithoutBufferedStream	0	10485760	10230
WRITE	ByteByByteWithoutBufferedStream	0	10485760	16512	READ	ByteByByteWithoutBufferedStream	0	10485760	10306
WRITE	ByteByByteWithoutBufferedStream	0	10485760	16480	READ	ByteByByteWithoutBufferedStream	0	10485760	10159





## Analyse des mesures

Avec ces différentes mesures, nous remarquons bien sûr que l'utilisation de tampons permet d'améliorer significativement les temps d'écriture et de lecture des données. Que ce soit avec ou sans tampons, nous remarquons également que l'utilisation de blocs de données de plus en plus grands (en écriture et en lecture), nous donne des performances de plus en plus grandes. Si nous comparons l'écriture et la lecture des données, il y a une plus forte tendance à ce que les temps se rapprochent en lecture avec ou sans utilisation de tampons. De plus, nous pouvons dire que la lecture des données est légèrement plus efficace en termes de temps que l'écriture de données avec et sans tampons. Enfin, nous remarquons que l'utilisation de tampons même avec une taille de bloc égale à 0 donne des temps bien plus petits que sans l'utilisation de tampons que ce soit en écriture ou en lecture.

## Modification du code du programme de base

Le programme qui nous était fourni au début du laboratoire ne permettait pas encore d'enregistrer les différents temps obtenus lors de l'expérience dans un fichier au format « csv ». Nous avons dû le modifier pour pouvoir le faire. Nous avons donc rajouté trois interfaces supplémentaires ainsi que trois classes implémentant ces trois interfaces. La première classe « ExperimentData » permet d'enregistrer les données que nous voulons écrire dans le fichier « csv » (comme le temps, la taille du fichier, la stratégie, ...) dans une map (avec une association entre une clé et une valeur). Ces données correspondent donc aux données générées durant l'expérience. La deuxième classe « FileRecorder » permet de créer le fichier de sortie au format « csv » ainsi que le flux de sortie (avec tampon) connecté au fichier de sortie. Enfin, la troisième classe « CsvSerializer » permet de sérialiser les données de l'expérience (récupérées depuis l'objet de la première classe) au format « csv » et de les écrire dans notre fichier de sortie. Le format « csv » correspond aux différentes données séparées par des « , ». Après avoir créé ces trois classes, nous les avons utilisées dans la classe principale du programme. Dans cette dernière, nous avons rajouté un constructeur dans lequel nous initialisons un objet pour la sérialisation (troisième classe) et un autre pour créer le fichier « csv » voulu (deuxième classe). Finalement, nous créons un nouvel objet de données (première classe) à chaque fois que nous appelons la méthode pour écrire des données de test dans des fichiers et celle pour lire ces mêmes données dans ces mêmes fichiers. Ci-dessous, le code permettant d'enregistrer les données de l'expérience dans un fichier « csv » est présenté (pour l'écriture de données).

```
// add data to the data object
data.addData("WRITE", ioStrategy.toString(), blockSize, NUMBER_OF_BYTES_TO_WRITE, time);
// record data added in the csv file
recorder.record(data);
```